A 'Cool' Way of Improving the Reliability of HPC Machines

Osman Sarood
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
sarood1@illinois.edu

Esteban Meneses
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
emenese2@illinois.edu

Laxmikant V. Kale
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kale@illinois.edu

ABSTRACT

Soaring energy consumption, accompanied by declining reliability, together loom as the biggest hurdles for the next generation of supercomputers. Recent reports have expressed concern that reliability at exascale level could degrade to the point where failures become a norm rather than an exception. HPC researchers are focusing on improving existing fault tolerance protocols to address these concerns. Research on improving hardware reliability, i.e., machine component reliability, has also been making progress independently. In this paper, we try to bridge this gap and explore the potential of combining both software and hardware aspects towards improving reliability of HPC machines. Fault rates are known to double for every 10°C rise in core temperature. We leverage this notion to experimentally demonstrate the potential of restraining core temperatures and load balancing to achieve two-fold benefits: improving reliability of parallel machines and reducing total execution time required by applications. Our experimental results show that we can improve the reliability of a machine by a factor of 2.3 and reduce the execution time by 12%. In addition, our scheme can also reduce machine energy consumption by as much as 25%. For a 350K socket machine, regular checkpoint/restart fails to make progress (less than 1% efficiency), whereas our validated model predicts an efficiency of 20% by improving the machine reliability by a factor of up to 2.29.

Keywords

Energy minimization, Temperature thresholds, Temperature capping, Fault tolerance, Thermal control, Checkpointing restart, Actionable modeling, Load balancing

1. INTRODUCTION

HPC research and its endeavor to build larger machines faces two major challenges today: power consumption and reliability. Estimates show that the combined energy consumption for all the data centers in the world is equivalent to 235 billion KWh that accounts for more than 1.3% of the world's overall electricity consumption [1]. Half of the energy consumed by a data center can be attributed to cooling [2, 3, 4]. Although the machine room temperature should be between 18°C-27°C as recommended by ASHRAE [5], most data centers operate at machine room temperatures in the range of 13°C-18°C [6] due to fear of increased node failures at higher temperatures. If processor operation at acceptable temperatures could be ensured, data center operators can run data centers at higher machine room temperatures. This is what motivates the temperature restraining component of our work.

Modern day microprocessors contain on-chip temperature sensors accessible through software with minimal overhead. These processors also provide means to change the voltage and frequency at which the chip operates, known as *Dynamic Voltage and Frequency Scaling* (DVFS). Running processors at lower voltage and frequency results in a reduction in power consumption. It also makes the processor cool down because of reduced thermal energy dissipation.

These qualities make DVFS very appropriate for keeping processors cool. A component of the runtime system can periodically check processor temperatures and decrease the frequency when the processor heats up beyond a user-defined threshold. If the temperature falls below a lower threshold, the runtime system can increase the processor frequency to improve performance. This scheme ensures that processor temperatures are restrained between a temperature range and avoids overheating. However, in tightly coupled science and engineering HPC applications, DVFS is not as straightforward because computations on one processor may be dependent on data produced by other processors. Therefore, if one processor slows down, it might cause the entire computation to slow down. Our earlier work [7, 8] experimentally showed that a temperature aware load balancer can be used in conjunction with DVFS-based temperature control to reduce cooling energy by as much as 57%. In addition to reducing cooling energy consumption, lower processor temperatures can also improve the reliability of a machine. Past work shows that the failure rate of a compute node doubles with every 10°C increase in temperature [9, 10, 11, 12].

In this work we show that by restraining processor temperatures, we can empower the user to *select* the reliability of the system from within a range. An increase in reliability can improve the performance of an application especially by using load balancing for overdecomposed systems [13]. We also show how different applications can affect the Mean Time Between Failures (MTBF) for a machine due to dif-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC '13 November 17-21, 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503228

ferent thermal profiles. We present and analyze the trade-offs of improving reliability and its associated cost, i.e the slowdown caused by DVFS-driven temperature control. In particular, this paper makes the following contributions:

- We analyze how restraining temperature of individual processors improves the reliability of the entire machine (§ 2.1).
- We formulate a model that relates total execution time of an application to reliability and the associated slowdown for temperature restraint (§ 2.2).
- We propose, implement and evaluate a novel approach that extends our earlier work [7, 8] and combines temperature restraint, load balancing and checkpoint/restart to increase reliability while reducing total execution time for an application (§ 3). We do several experiments that span over an hour and have at least 40 faults. This work is, as far as we know, the first extensive experimental study that provides insights on the effects of temperature restraint on estimated $MTBF$ for HPC machines.
- We first validate the accuracy of our model (§ 4) and then use it to show the scheme's expected benefits for larger machines (§ 5). Our results show that for a 340K socket machine, we improve the machine efficiency from 0.01 to 0.22 as a result of improving the machine reliability by a factor of up to 2.29.

2. IMPLICATIONS OF TEMPERATURE CONTROL

Processor temperature has a profound impact on the fault rate of a processor. For every 10°C increase in processor temperature the fault rate doubles [9, 10, 11, 12]. We refer to this this relation as the *10-degree* rule in the rest of the paper. While restraining processor temperature improves reliability, it also causes an execution time slowdown due to DVFS. In this section, we use temperature control to estimate the improvement in reliability and its impact on the total execution time of an application.

2.1 Effects of temperature control on reliability

MTBF for a processor (m) is exponentially related to its temperature and can be expressed as: [12, 14, 15]

$$m = A * e^{-b*T} \quad (1)$$

where T is the processor temperature, A and b are constants. Assuming an m of 10 years at 40°C, m per processor based on the 10-degree rule can be expressed as:

$$m = 160 * e^{-0.069T} \quad (2)$$

In a system where the failure of a single component can cause the entire application to fail, the MTBF of the system can be defined as (M) [16]:

$$M = \frac{1}{\sum_{n=1}^{N} \frac{1}{m_n}} \quad (3)$$

where N is the number of nodes and m_n is the MTBF for socket n. Although the absolute value of core temperatures is important for each processor's reliability (Equation 2), reliability of the entire cluster also depends on the variance

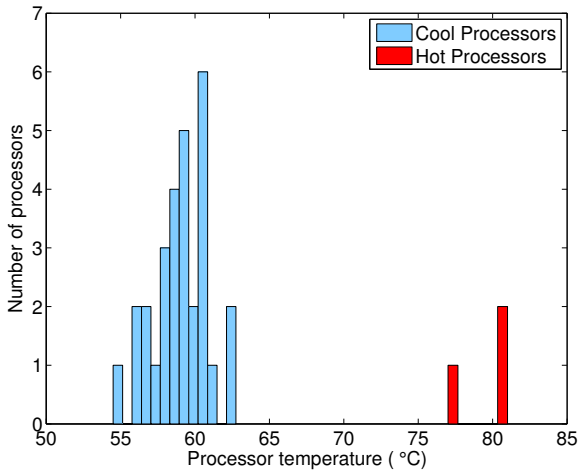


Figure 1: Histogram of max temperature for each node of the cluster using *Wave2D*

of core temperatures for all the processors present in the cluster (Equation 3). Presence of hot spots can degrade reliability of the system.

To analyze processor temperature behavior, we ran a 5-point stencil application, *Wave2D*, on a 32 node (128 cores) cluster for over 30 mins and recorded the maximum temperature reached by each processor. The results are pictured in Figure 1 where each bar shows the number of processors reaching a specific maximum temperature during the 30-min run. The red bars in Figure 1 indicate the presence of a hot spot composed of three processors that heated up to 78°C-80°C. The maximum temperature reached by the remaining 29 processors ranged from 55°C-63°C (shown in blue). The average temperature for the cool processors was $T_c = 59$°C, with a standard deviation of $\sigma = 2.17$°C. Feeding the temperature data from Figure 1 to Equations 2 and 3 estimates the MTBF to be 24 days for our cluster. As Equation 2 outlines, we can increase m for each processor by restraining its temperature to a lower value and hence increase overall M for the cluster. To estimate the improvement in M, we do the following:

1. Remove the hot spot by bringing the *hot* processors' distribution back to that of *cool* processors
2. Shift the entire distribution towards the left so that the processors operate at an average temperature of 50°C instead of 59°C

Suppose we remove the hot spot by restraining temperature for the three hot processors to $T_c = 59$°C, i.e., the average temperature for cool processors. Using these new temperature values for the three processors in the hot spot, along with the actual temperature values we got for cool processors, we re-estimate M and notice an increase of 7 days (from 24 to 31 days). The estimated improvement in M after hot spot removal is shown by the dashed line in Figure 2 which joins the two points representing value of M with (red circle) and without (blue diamond) hot spot. We can now predict M given a temperature restraint for a processor in the hot

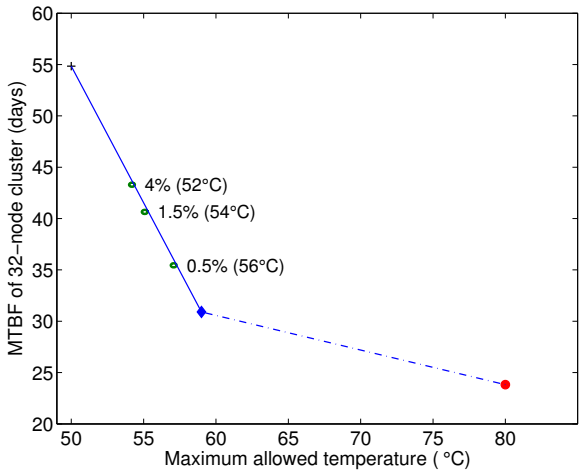


Figure 2: Effect of cooling down processors on MTBF of the system

spot. For example, keeping the three processors in the hot spot to 70°C would result in an estimated 27.5 days for M.

So far we've established that hot spot removal improves M. Next, we study the effect of restraining temperatures for all the processors to 50°C. For this, we generate 32 normally distributed random temperature values with a mean of 50°C and $\sigma = 2.17$°C (same as cool processors) and re-estimate M. The improvement in M for any temperature restraint between 50°C to 59°C (black '+' and blue diamond in Figure 2 respectively) can be estimated from the solid line in Figure 2.

These improvements motivated us to use our temperature aware load balancer [8] for restraining temperatures and for studying the slowdown associated with using DVFS for temperature control. For the purpose of this study, we assume that the reliability stays constant while the input voltage to the processor is decreased. To test how well DVFS restrains core temperature, we ran *Wave2D* using different temperature thresholds. The small green dots in Figure 2 represent experiments carried out on our 32-node cluster. The percentage in the labels represents the slowdown in execution time compared to the experiment where temperatures are not restrained and all processors always work at maximum frequency. The number in brackets shows the temperature threshold used for that experiment. Decreasing temperature threshold causes the points to move towards the left indicating a decrease in average temperature for all the processors. Since the processors are operating at lower temperatures, the estimated M keeps increasing according to Equations 2 and 3. However, this improved reliability comes at the cost of DVFS induced slowdown which keeps increasing with reduction in temperature threshold.

2.2 Effects of temperature control on total execution time

In this section, we focus on analyzing whether improvement in M is significant enough to overcome the slow down associated with temperature control. To this end, we combine the checkpointing technique for fault tolerance from [17] with temperature control, to formulate the resulting execution time. This formulation will allow us to investigate the relative impact of different parameters of our framework and enable us to project the results to exascale.

Checkpoint-restart mechanism saves the current state of an application for later restart. Checkpoint time (δ), is the time to dump application state to local storage and checkpoint period (τ) is the frequency of checkpointing. In a fault prone environment, if the system checkpoints too often, then time may be wasted unnecessarily in dumping the checkpoints. In contrast, a low checkpoint frequency will mean a high amount of work lost in a failure and large recovery time. Therefore, it is clear that a balance must be found. There are well-known models [18, 19] to determine the optimum checkpoint period for a particular combination of system and application.

We leverage DVFS, to incorporate temperature control and its corresponding slowdown, to extend a popular checkpoint/ restart model [18]. Our model assumes that failure arrival is exponentially distributed and failures are independent of each other. We use a collection of parameters to represent different factors that affect performance of a resilient framework. Table 1 lists all the parameters of our performance model along with a short description of each.

Table 1: Parameters of the performance model

Parameter	Description
W	Time to completion in a fault-free scenario
M	MTBF of the system
T	Total execution time
δ	Checkpoint time
τ	Optimum checkpoint period
R	Restart time
μ	Temperature control slowdown

With the above parameters, we obtain the total execution time of an application as follows:

$$T = T_{Solve} + T_{Checkpoint} + T_{Recover} + T_{Restart} \quad (4)$$

where T_{Solve} is the time it takes to complete program execution in fault-free scenario, $T_{Checkpoint}$ is the total checkpointing time during the entire program execution, $T_{Recover}$ is the time to recover lost work for all the faults occurring during execution, and $T_{Restart}$ is the time necessary to detect all the failures and have the entire system ready to resume execution.

The detailed formulation for total execution time (T) of a program under temperature restraint becomes:

$$T = W\mu + \left(\frac{W\mu}{\tau} - 1\right)\delta + \frac{T}{M}\left(\frac{\tau+\delta}{2}\right) + \frac{T}{M}R \quad (5)$$

here μ is the ratio between an application's total execution time in a fault-free scenario with and without temperature restraint. In other words, the parameter μ represents the cost of temperature restraint that includes load balancing decision time as well as object migration. In Equation 5, $\left(\frac{W\mu}{\tau} - 1\right)$ represents the number of checkpoints, $\frac{T}{M}$ is the number of faults expected to occur during execution, and $\left(\frac{\tau+\delta}{2}\right)$ is the average recovery time per fault.

3. APPROACH

In this section, we propose a novel approach, based on task migration and temperature control, to efficiently *control* the estimated reliability of HPC machines (within a range). While doing so, our approach simultaneously minimizes total execution time including the overheads of fault tolerance, i.e., checkpointing, recovery and restart. Our scheme should work well with any parallel programming framework allowing task migration. We start by giving an overview of the system model, followed by details of how to use DVFS and task migration to efficiently restrain processor temperature. We then discuss the checkpoint/restart mechanism and conclude by giving an overview of how to combine temperature control, load balancing and checkpoint/restart.

3.1 System Model

We conceive the underlying machine as a set of *processors* connected through a network that does not guarantee in-order delivery. Each processor is able to run an arbitrary number of *tasks*. The collection of all the tasks running on the processors compose the parallel application. Each task will hold a portion of the data and perform its part of computation. The only mechanism to exchange information in the task set is via message passing.

Tasks are *migratable*: each task can serialize its state and be moved to a different processor. A smart runtime system is responsible for monitoring the underlying machine and balancing the load of different processors to achieve better performance. The runtime system uses synchronization points in the application to trigger load balancing and checkpoint/restart frameworks. The runtime system also monitors the temperature in each processor and can change the frequency at which processors operate.

3.2 Temperature control and communication aware load balancer

We now describe our temperature control mechanism along with communication aware load balancing to mitigate the cost of temperature restraint. The idea is to let each processor work at the maximum possible frequency as long as it is below a user-defined maximum temperature threshold. Since machines of today do not allow DVFS on a per-core basis, we use the average temperature for all the on-chip cores to decide whether or not to change the frequency. A key parameter for us is the lower temperature threshold after which we can increase the frequency of the chip. If this lower threshold is close to the maximum threshold, it can cause *frequency flapping* and lead to expensive object migrations done to achieve load balance.

The pseudocode for our temperature restrain strategy is given in Algorithm 1 with a description of variables in Table 2. We start with all the processors checking their temperature against the user defined maximum threshold. If the temperature (t_i^k) exceeds the max threshold, the frequency for that chip (C_i) is decreased by one level (P-state). In contrast, if the temperature is less than T_{min}, operating frequency for that chip is increased by one level.

Once the frequencies have been changed, the system might become load imbalanced where some processors (with lowered frequency) are now overloaded. We leverage task migratability to correct the load imbalance and transfer objects from the slower-hot processors to the faster-cool processors. This load balancing strategy is an extension of our previous work [8] which did not account for communication costs in its load balancing decisions. Algorithm 2 shows the pseudocode for our *communication aware load balancer*. We start by estimating the total *ticks* required for each task during the last load balancing period as a product of each task's execution time and the frequency at which its *host* processor was operating (line 3). To fix load imbalance, we calculate the amount of work assigned to each processor (*procTicks*) during the recent load balancing period in terms of ticks (line 7). While calculating *procTicks*, we also calculate the sum of frequencies (*sumFreqs* at line 8) at which all the processors should operate in the *coming* load balance period. We use *sumFreqs* to categorize a processor as *heavy* or *light* for the upcoming load balance period in the function *createOverHeapAndUnderSet*. This function takes the *procTicks* for all the processors and uses *isHeavy* and *isLight* methods (line 26-line 30) to determine if a processor is overloaded or underloaded based on a *tolerance* number. It uses the *isHeavy* method to create a max heap for all the overloaded processors whereas the underloaded processors are determined by using *isLight* method and are kept in a set.

After identifying overloaded and underloaded processors, we transfer tasks from the former to the latter until no overloaded processors are left in the max heap (line 11-line 24). For migration cost minimization, we assume that the initial task-to-processor mapping (m vector) is the best and strive to restore it when trying to transfer tasks. To track the initial mapping, we introduce the notion of a $foreign$ task. A task is said to be $foreign$ if it currently resides on a processor other than the one to which it was initially mapped. We then pop the most overloaded processor from the max heap (line 12) and check if it has any $foreign$ tasks (line 13). If so, we randomly select one foregin task (line 14), otherwise we randomly select one regular task (line 16). Once the $bestTask$ is determined, we look for the best possible

Table 2: Description for variables used in Algorithm 1 and Algorithm 2

Variable	Description
n	number of tasks in the application
p	number of processors
T_{max}	maximum temperature allowed
T_{min}	minimum temperature allowed
k	current load balancing step
$taskTime_i^k$	execution time of task i during step k (in ms)
$procTime_i^k$	time spent by processor i executing tasks during step k (in ms)
f_i^k	frequency of processor i during step k (in Hz)
m_i^k	processor number assigned to task i during step k
$taskTicks_i^k$	number of clock ticks taken by i^{th} task during step k
$procTicks_i^k$	number of clock ticks taken by i^{th} processor during step k
t_i^k	average temperature of chip i at start of step k (in $^\circ$C)
$overHeap$	heap of overloaded processors
$underSet$	set of underloaded processors

Algorithm 1 Temperature Control

1: On every processor i at start of step k
2: **if** $t_i^k > T_{max}$ **then**
3: decreaseOneLevel(C_i) {increase P-state}
4: **else if** $t_i^k < T_{min}$ **then**
5: increaseOneLevel(C_i) {decrease P-state}
6: **end if**

processor to transfer it to. The function *getBestProcList* (line 31-line 36) takes the *bestTask*, iterates over all underloaded processors and calculates the amount of communication that occurs between the *bestTask* and each of the underloaded processors i. The function *getCommForTask* on line 33 takes the *bestTask* along with an underloaded processor i and returns the amount of communication that occurs between them in Kilobytes. Using the candidate processors from *sortedProcsList* (line 18), the method *getBestProc* selects the processor (*bestProc*) which communicates the most with *bestTask* and would not be overloaded after receiving *bestTask*. To trigger the actual transfer, the mapping ($m^k_{bestTask}$) is updated along with the *procTicks* variables for both the *donor* and the *bestProc* (receiver) at line 20-22. Now that the *bestTask* has been decided for migration from *donor* to *bestProc*, we update the loads of *overHeap* and *underSet* to reflect this migration (line 23) and continue the loop from line 11.

3.3 Checkpoint/Restart

Rollback-recovery techniques are highly popular in large-scale systems to provide fault tolerance. Among those techniques, checkpoint/restart is the preferred mechanism in HPC. The fundamental principle behind checkpoint/restart is to periodically save the state of the system and rollback to the latest checkpoint in case of a failure. Several libraries implement one of the many variants of checkpoint/restart [17, 20, 21, 22].

Our fault tolerance scheme is called double local-storage checkpoint/restart [17]. Local-storage refers to any storage device local to the processor (main memory, solid-state drive, local hard disk). Additionally, every processor will store a checkpoint copy in two places. One checkpoint copy will be saved in the local storage of the processor and another copy in the local storage of a checkpoint *buddy*. In case of a failure, all processors will rollback to the previous checkpoint. The affected processor will receive the checkpoint from its buddy. The rest of the processors will pull the checkpoint from their own local storage.

Checkpointing is performed in coordination such that all participating processors store their checkpoint at a synchronization point determined by the programmer. Once the checkpoint call is made, every processor collects the state of all the tasks residing on it and proceeds to store its two copies of the checkpoint. The runtime system provides a simple interface for each task to dump its state.

We assume the underlying system runs a failure detection mechanism with a processor being considered as the failure unit. Indeed, our checkpoint/restart is resilient to single-processor failures. Multiple-processor failures may be tolerated, but there is no total certainty in the general case. We follow the *fail-stop* model for processor failures. This means, after a processor crashes, it becomes unavailable and does not come back again. Such processor is replaced by a spare

Algorithm 2 Communication Aware Load Balancing

1: On Master processor
2: **for** $i \in [1, n]$ **do**
3: $taskTicks_i^{k-1} = taskTime_i^{k-1} \times f^{k-1}_{m_i^{k-1}}$
4: $totalTicks \mathrel{+}= taskTicks_i^{k-1}$
5: **end for**
6: **for** $i \in [1, p]$ **do**
7: $procTicks_i^{k-1} = procTime_i^{k-1} \times f_i^{k-1}$
8: $freqSum \mathrel{+}= f_i^k$
9: **end for**
10: createOverHeapAndUnderSet()
11: **while** *overHeap* NOT NULL **do**
12: $donor$ = deleteMaxHeap($overHeap$)
13: **if** $numForeignObjs(donor) > 0$ **then**
14: $bestTask = getForeignTask(donor)$
15: **else**
16: $bestTask = getRandomTask(donor)$
17: **end if**
18: $sortedProcsList$ = getBestProcsList($bestTask$)
19: $bestProc$ = getBestProc($sortedProcsList$)
20: $m^k_{bestTask} = bestProc$
21: $procTicks^{k-1}_{donor} \mathrel{-}= taskTicks^{k-1}_{bestTask}$
22: $procTicks^{k-1}_{bestProc} \mathrel{+}= taskTicks^{k-1}_{bestTask}$
23: updateHeapAndSet()
24: **end while**
25:
26: **procedure isHeavy(i)**
27: return $procTicks_i^{k-1} > (1 + tolerance) * totalTicks * (f_i^k / freqSum)$
28:
29: **procedure isLight(i)**
30: return $procTicks_i^{k-1} < totalTicks * f_i^k / freqSum$

31: **procedure getBestProcList(**$bestTask$**)**
32: **for** $i \in underSet$ **do**
33: $bestProcs[i].comm$ = getCommForTask(i,$bestTask$)
34: $bestProcs[i].procId$ = i
35: **end for**
36: return $bestProcs$

processor taken from a pool of available processors.

3.4 Framework

In this section, we explain how we provide controllable resilience for HPC systems by bringing together all three modules of our approach, i.e., temperature control (TC), communication aware load balancing (LB) and checkpoint-restart. Figure 3 shows our framework with a system of two processors (X and Y) running a total of five tasks (from A to E) that are executed in each iteration of a parallel program. The initial distribution of tasks places tasks A and B on processor X while tasks C, D, and E are mapped to processor Y.

As Figure 3 shows, the program performs temperature control and load balancing (TC & LB) several times during a checkpointing period, i.e., between two adjacent checkpoints. The program performs several iterations until the TC & LB modules are called after iteration i. The TC module detects processor Y running at a temperature higher than the max threshold and reduces its frequency. Following this, the LB module takes control and removes the load

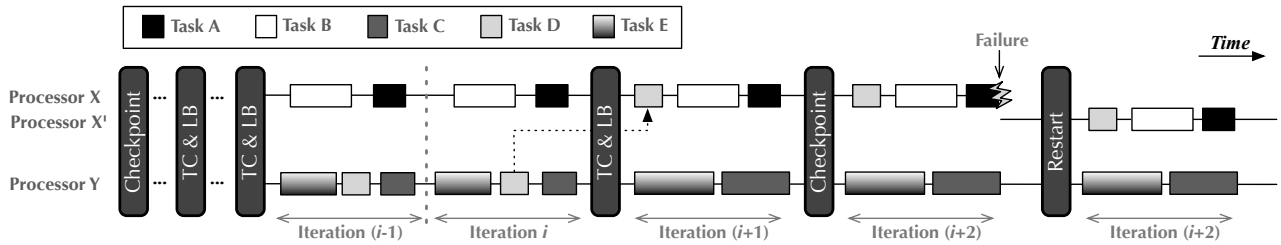


Figure 3: Dynamic power management and resilience framework. The runtime system routinely adjusts the frequency of processors and solves the load imbalance that may appear. This temperature-capping process decreases the failure rate. However, if a failure occurs, the checkpoint/restart mechanism provides fault tolerance.

imbalance by migrating task D from Y to X as outlined in Algorithm 2. The system checkpoints after iteration $i+1$ and continues with execution. A failure takes down processor X during iteration $i+2$, which gets replaced by a spare processor that we call processor X'. The checkpoint buddy of processor X provides checkpoint data for X to the replacement processor X' to resume execution until the program finishes.

4. EXPERIMENTS

In this section, we provide a comprehensive experimental evaluation of our techniques using three different applications. The first one is *Jacobi2D*: a canonical benchmark that iteratively applies a five-point stencil over a 2D grid of points. The second application, *Wave2D*, uses a finite difference scheme over a 2D discretized grid to calculate the pressure resulting from an initial set of perturbations. The third application, *Lulesh*, is a shock hydrodynamics application which was defined and implemented by Lawrence Livermore National Laboratory (LLNL) [23].

The rest of this section describes our implementation, testbed and experimental results. All experimental results are based on real hardware, and there are no simulation results in this section.

4.1 Implementation using Charm++

CHARM++ is a parallel programming runtime system that leverages processor virtualization. It provides a methodology where the programmer divides the program into smaller chunks (objects or tasks) which are distributed amongst the **p** available processors by the runtime system [13]. Each of these small chunks is a migratable C++ object that can reside on any processor. The runtime system keeps track of task execution time and maintains this log in a database to be used by a load balancer for quantifying the amount of work in each task [24].

Based on this information, the load balancer in the runtime system detects load imbalance following which it migrates objects from an overloaded processor to an underloaded one [25]. The load balancing decision is based on the heuristic of *principle of persistence*, according to which computation and communication loads tend to persist over time for a certain class of iterative applications. At small scales, the cost of the entire load balancing process, from instrumentation through migration, is generally a small portion of the total execution time, and less than the improvement it provides in execution time. When this does not hold, a strategy must be chosen or adapted to match the application's needs [26]. Our communication aware load balancer can be easily adapted to existing hierarchical schemes, which have been shown to scale to the largest machines available [27]. By limiting the cost of decision-making and scope of migration, we expect these schemes to offer similar benefits while restraining processor temperatures.

4.2 Testbed and experimental settings

We evaluated our scheme on a cluster of 32 nodes (128 cores). Each node has a single socket with a four-core Intel Xeon X3430 processor chip. Each chip can be set to 10 different frequency levels ('P-states') between 1.2 GHz and 2.4 GHz. It also supports Intel's TurboBoost [28], allowing processors to overclock up to 2.8 GHz. The operating system on the nodes is CentOS 5.7 with `lm-sensors` and the `coretemp` modules installed to provide core temperature readings. We use the `cpufreq` module to enable software-controlled DVFS. The cluster nodes are connected by a 48-port gigabit ethernet switch. We use a Liebert power distribution unit installed on the rack containing the cluster to measure the machine power at 1 second intervals on a per-node basis. We gather these readings for each experiment and integrate them over the execution time to obtain the total machine energy consumption.

In Section 2.1, we estimated the $MTBF$ for our cluster (M) to be in the range of 24 - 55 days. To carry out experiments representative of a much larger system, we scale our cluster M proportionally. We chose an m of 1 hour at 40°C per socket. For a system of 690K sockets, these settings emulate an m of 10 years per socket. After demonstrating the accuracy of our model by showing that it closely matches experimental results, we make predictions for larger machines. The three applications we considered exhibited different temperature profiles. Therefore, to make our experiments realistic, we used actual temperature values to estimate M for each application for experiments without temperature restraint. More precisely, we estimate M using the max temperature that each of the 32 nodes reaches. Table 3 shows the cluster-wide average max temperatures for each application in case of no temperature restraint. We refer to this baseline case as NC for the rest of the paper. For experiments where we restrain temperatures, we use the maximum temperature threshold to estimate M. The values of M corresponding to each temperature threshold are shown in Table 4. After determining M for each temperature threshold, we generate sequences of exponential random numbers for each experiment, by taking each M as the distribution mean. We manually insert faults according to these random number sequences for each experiment, by killing a process on any one of the nodes using the `kill -9` command

Table 3: Application parameters for NC case

Parameter	Lulesh	Jacobi2D	Wave2D
δ (s)	9.57	7.65	8.01
T_{avg}(°C)	55.31	53.42	55.56
M (s)	40.31	44.40	39.02
τ(s)	18.2	18.4	17.0
R (s)	2.2	1.52	1.60
Recovery(%)	33.31	29.05	31.19
Checkpointing(%)	21.40	20.11	20.89
Restart(%)	5.38	3.48	4.03

to wipe off all data. To recover from the artificially inserted failures, we calculate the optimum checkpoint period (τ) for each experiment as follows [18]:

$$\tau = \sqrt{2\delta M} - \delta \quad (6)$$

Given that τ depends on M and the checkpoint time (δ), we obtain a different optimum checkpoint period for each application when running at a given temperature threshold. The δ for each application is listed in Table 3.

4.3 Experimental results

To establish and quantify the benefits of our scheme and to validate the accuracy of our model (outlined in Section 2.2), we carried out a number of experiments. We demonstrate how we can improve reliability using temperature control and compare the execution time for experiments with and without temperature control. All experiments reported in this section are compared to the baseline experiments (represented by NC) where all processors always operate at the max frequency without any temperature control. Since the load balancing technique is not the main focus of this study, we would not be giving a detailed comparison of the improved load balancer proposed in Section 3 and the one proposed in our earlier work [8]. However, our proposed new strategy does improve execution time for all three applications. For T_{max} =49°C, the communication-aware load-balancer can reduce execution time by 14%, 18%, and 5% for *Wave2D*, *Lulesh* and *Jacobi2D*, respectively, compared to the load-balancer proposed in our earlier work [8]. Each data point (experiment) reported in this section represents a benchmark running for more than 1 hour and being subject to at least 40 faults.

Table 3 lists the average temperature for each application. Both *Lulesh* and *Wave2D* have an average max temperature that is 2°C higher than that for *Jacobi2D*. Due to this difference in temperature profile, we ran *Jacobi2D* for a max temperature threshold range of 42°C-52°C as opposed to 44°C-54°C used for *Lulesh* and *Wave2D*. This difference in thermal profile is also responsible for making different applications operate at different average frequencies. For example, when running below a temperature threshold of 46°C, the average frequencies across the cluster for *Lulesh*, *Jacobi2D* and *Wave2D*, were 2.30 Ghz, 2.31GHz and 2.27 GHz, respectively. Although our testbed has a maximum Turbo Boost frequency of 2.8Ghz, using DVFS to restrain temperatures resulted in lower average frequency for all the applications. A detailed discussion about the interaction between temperature, frequency and performance can be found in our earlier work [8].

Table 4: MTBF (sec) for different temperature thresholds (42°C - 54°C)

T_{max}	54	52	50	48	46	44	42
M	43.8	50.4	57.8	66.4	76.2	87.5	100.2

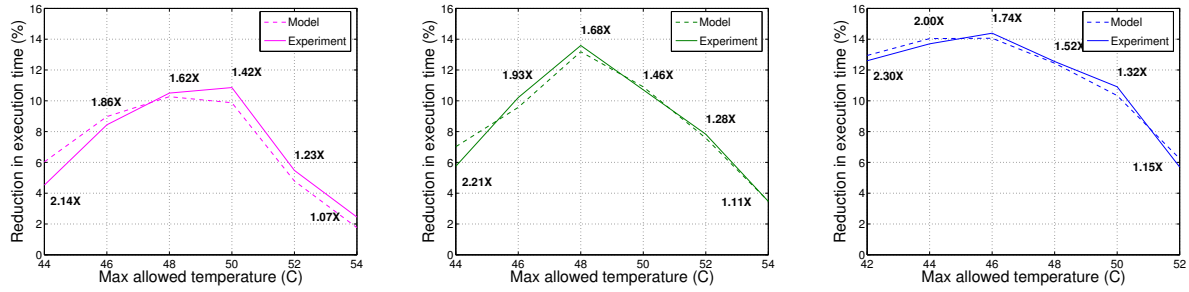
Figure 4 shows percentage reduction in execution time using both temperature restrain and load balancing compared to the baseline experiments, i.e., NC. The two curves in each plot compare experimental results with model predictions, both of which closely match. The model predictions for Figure 4 were gathered by feeding checkpoint time, slowdown, restart time and useful work time to Equation 5 and using golden section search and parabolic interpolation to optimize τ for minimum total execution time. It is not surprising to see the inverted U shape of all three curves which strongly suggests a tradeoff between reliability (M) and the DVFS induced slowdown (μ) due to temperature restraint. Figure 4 also shows the ratio of M for the machine using our scheme relative to the NC case. For example, by restraining temperatures to 42°C in case of *Jacobi2D*, M for the machine increased 2.3 times compared to the case of NC. Hence, restraining the temperature to a lower value may decrease the benefits of our scheme but it would always improve estimated reliability of the machine.

4.3.1 Interplay between temperature, MTBF and checkpointing overheads

$MTBF$ for a machine (M) is dependent on each processor's temperature. Higher processor temperatures for *Lulesh* and *Wave2D* imply a lower M than *Jacobi2D* (Table 3). This forces *Wave2D* and *Lulesh* to spend a higher percentage of time in recovery as they encounter more failures compared to *Jacobi2D* (Table 3). Although M for *Wave2D* and *Lulesh* are very close in case of NC, they spend different percentage of time in recovery, i.e., 31.19% and 33.31% respectively. This observation can be explained by looking at the τ values for both of them (Table 3). According to Equation 6, a larger checkpoint time (δ in Table 3) for *Lulesh* results in a larger τ which increases the *average* recovery time for *Lulesh* ($\frac{\tau+\delta}{2}$ in Equation 5). On the other hand, *Lulesh*'s higher τ causes it to spend almost an equal % of time in checkpointing as *Jacobi2D* and *Wave2D* , i.e., 21.40% (Table 3), despite *Lulesh*'s large δ. Although the time per checkpoint (δ) for *Lulesh* is the highest, the product of number of checkpoints and δ comes out to be equal to other applications due to fewer checkpoints $\left(\frac{W\mu}{\tau} - 1\right)$ for *Lulesh*. *Lulesh* also spends 5.38% of total time in restarts which can be attributed to the higher restart cost of 2.2 secs (Table 3).

4.3.2 Comparing the benefits amongst applications

Although all three applications have a inverted U shaped curve, their maxima occur at different temperature thresholds. We define this optimum point for each application by the tuple (T_{max}, r_{max}), where T_{max} is the temperature threshold corresponding to the point representing maximum reduction in execution time for an application. Figure 4 shows that the optimum points for *Jacobi2D*, *Wave2D*, and *Lulesh* are (46°C,14.2%), (48°C,13.5%), and (50°C,11%) respectively. We notice that the applications differ in *both* members of the tuple. An application's optimum point de-



(a) Lulesh (b) Wave2D (c) Jacobi2D

Figure 4: Reduction in execution time for different temperature thresholds

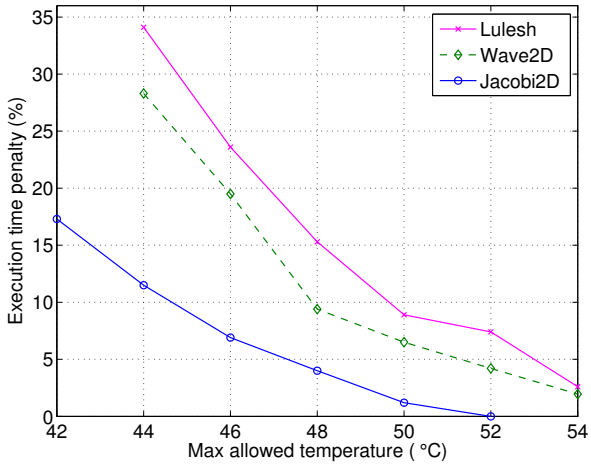


Figure 5: Execution time penalty for DVFS

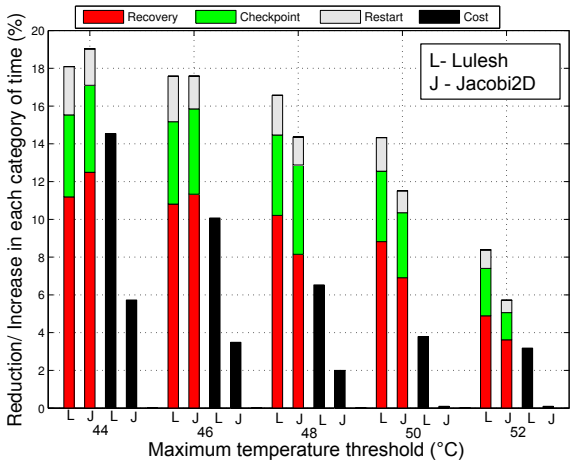


Figure 6: Gains/cost of increasing reliability for different temperature thresholds

pends on the tradeoff between percentage reduction in each category of total time (recovery, checkpoint and restarting times) which is a result of improvement in M, and its associated cost. This slowdown, including overhead of object migration during load balancing, is shown in Figure 5.

Another observation we can make is that the temperature threshold and the cost of temperature control μ are directly related. Figure 5 shows that *Lulesh* had the maximum slowdown leading to a larger optimum temperature threshold (50°C) and therefore it receives minimum reduction in execution time (11%) among all three applications. On the other hand, *Jacobi2D*, experiences the least slowdown which results in the maximum reduction in execution time, i.e., 14.2%, and the lowest optimum temperature threshold. The slowdown for *Wave2D* lies in between *Lulesh* and *Jacobi2D* which results in a reduction in execution time that is between 11%-14.2%, i.e., 13.5%.

4.3.3 *Understanding application response to temperature restraint*

All applications we considered respond differently to temperature restraints which is why each one has a different optimum point. For more insights, we compare and contrast how *Jacobi2D* and *Lulesh* respond to temperature control in Figure 6. Here, we plot the percentage of time reduced for each category of execution time (including recovery, checkpoint, and restart times), as a percentage of total time taken in case of NC.

We used the following formula to determine the recovery percentage (p_i^{rec}) corresponding to the max temperature threshold of i°C for Figure 6:

$$p_i^{rec} = \frac{t_{NC}^{rec} - t_i^{rec}}{T_{NC}} * 100 \quad (7)$$

where T_{NC} is the total execution time in case of NC, t_{NC}^{rec} is the recovery time for NC and t_i^{rec} is the recovery time for the experiment where the max threshold was i°C. Figure 6 also shows the cost of temperature control for *Lulesh* and *Jacobi2D* which represents DVFS-incurred slowdown in doing *useful work* ($W\mu$ in Equation 5). This cost p_i^{cost} as well as the percentage reduction in checkpoint p_i^{ckpt} and restart times p_i^{res} are calculated similar to p_i^{rec} in Equation 7. We make two observations from Figure 6.

First, we look at the total gain (sum of recovery, checkpointing and restart gains). While the total gains are always greater for *Lulesh* compared to *Jacobi2D* (except for 44°C),

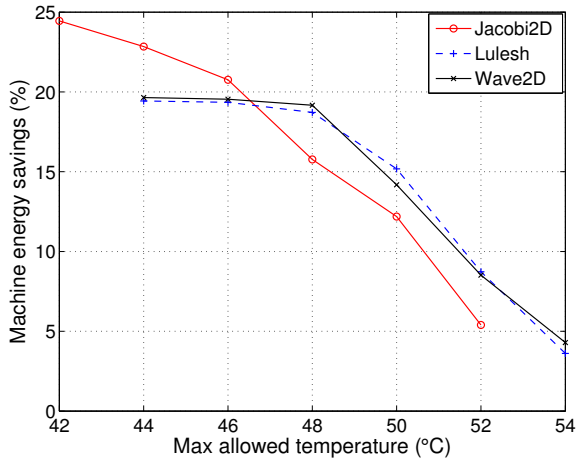


Figure 7: Reduction in machine energy consumption for all applications

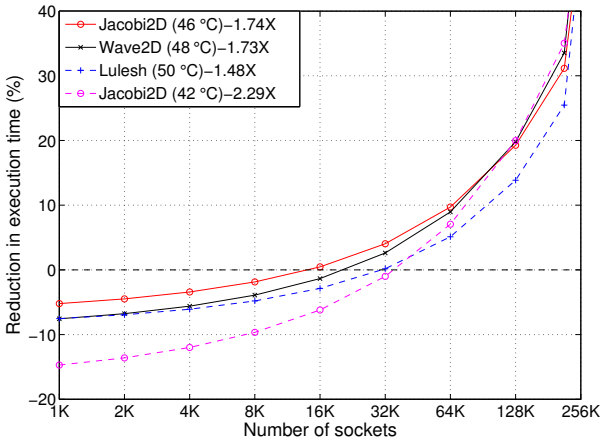


Figure 8: Execution time reduction for all applications at very large scale

its cost of temperature control is always significantly lesser than that for *Lulesh*. Hence, the net gain (*total gains - cost*) for *Jacobi2D* makes it much more appropriate for our scheme compared to *Lulesh*.

Next, we observe that p_{48}^{rec}, p_{50}^{rec} and p_{52}^{rec} are higher for *Lulesh* compared to *Jacobi2D* whereas for lower thresholds, p_{44}^{rec} and p_{46}^{rec} are lower for *Lulesh*. Recall from Figure 4 that *Lulesh* improves reliability of the system more than *Jacobi2D*, i.e., (1.86X, 2.14X) compared to (1.74X, 2.00X) for thresholds of 46°C and 44°C, respectively. Even then, the high timing penalty for *Lulesh* depicted in Figure 6 is limiting the gains from increased reliability. The timing penalty not only contributes directly as cost of improving reliability by prolonging useful work, it also indirectly affects the benefits of our scheme by limiting the gains we obtain in recovery. So if a timing penalty of μ gets added to the total execution time, then the faults, checkpoints and restart that happen during μ essentially work to cancel out some of the gains achieved by temperature restraint during the earlier part of execution. This is precisely why the timing penalty shrinks the gain bars in Figure 6. However, even with the higher timing penalty of *Lulesh*, its gains are sufficient to reduce execution time as compared to the case of NC.

4.3.4 Reduction in energy consumption

After highlighting how our scheme successfully reduces execution time and increases M, we now analyze the reduction in machine energy consumption that happens as a direct consequence of our scheme. Figure 7 shows the percentage reduction in machine energy consumption for each application compared to the baseline case (NC). These numbers represent actual machine energy consumption for experiments measured using power meters. The figure shows that we were able to reduce machine energy consumption by as much as 25% in case of *Jacobi2D* by restraining processor temperatures at 42°C. Although the reduction in execution time contributes to reduction in energy consumption, the major part of savings comes from temperature control which reduces a machine's power consumption. In addition to the reported reduction in machine energy, our scheme should also reduces the cooling energy significantly [8].

5. LARGE-SCALE PROJECTIONS

In Section 4, we thoroughly investigated our approach and validated our model by carefully comparing it against experimental results. Now, we use the validated model to project the benefits of our scheme for larger machines. We estimate improvement in machine efficiency for larger number of sockets and also analyze the benefits of our scheme while increasing memory size of an exascale machine.

5.1 Benefits for increasing number of sockets

Figure 8 shows the reduction in execution time we achieve for all three applications compared to the case of NC. For this plot, we use an m of 10 years per socket with a restart time of 30 secs. We show all three applications using their optimum temperature thresholds (T_{max}) from Section 4.3.2. Moreover, to highlight how T_{max} influences the reduction in execution time, we plot *Jacobi2D* for T_{max}= 42°C as well. We assume checkpoint time to be 240 secs [29]. The dashed black line in Figure 8 shows 0% reduction in execution time. The points below this signify an overhead of our scheme whereas the ones above this line represent reduction in total execution time using our scheme. The numbers in the legend of Figure 8 represent the *times* improvement in M for each application. Even though we can see an execution time penalty of 15% for 1K sockets in case of *Jacobi2D* with a T_{max} of 42°C, it increases the reliability of the machine by a factor of 2.29X. The same *Jacobi2D* runs with lesser penalty at T_{max} of 46°C for 1K sockets but its reliability decreases to 1.74X.

For a smaller number of sockets(less than 32K), running *Jacobi2D* with a T_{max} of 42°C incurs a cost that is much higher than the gains. However, beyond the crossover point (32K), the cost is justified as the gains become significantly higher. Hence, the optimum T_{max} can be different for different applications at various scales, e.g., at 230K sockets, *Jacobi2D* with a T_{max} of 42°C reduces the execution time by 38% compared to 32% if run at T_{max} of 46°C.

Efficiency can be defined as the fraction of the total ex-

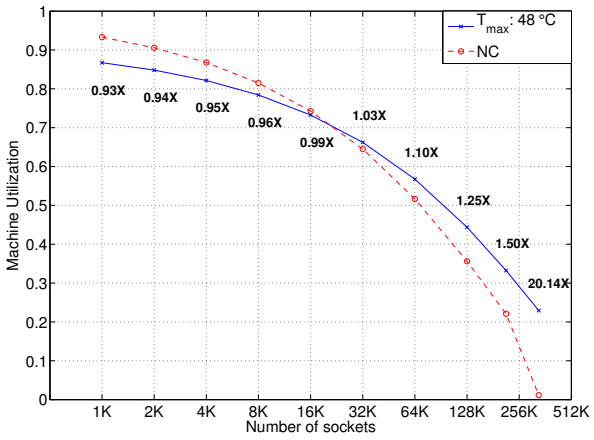


Figure 9: Projected efficiency for *Wave2D*

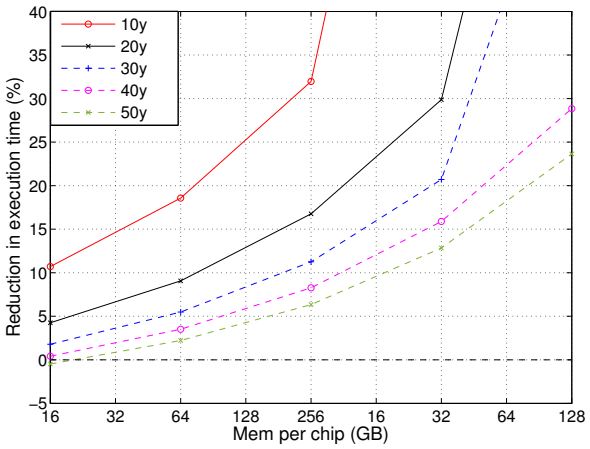


Figure 10: Reduction in execution time for different memory sizes of exascale machine

ecution time, including the fault tolerance overheads, that is spent in doing useful work. A decrease in total execution time can be thought of as an improvement in machine efficiency. Figure 9 plots the machine efficiency for *Wave2D* at T_{max} = 48°C using the same parameters as Figure 8. It is encouraging to observe that even though we account for DVFS incurred slowdown in our efficiency calculation, our scheme still improves machine efficiency significantly for larger number of sockets. The numbers shown in Figure 9 represent the ratio of efficiency for our scheme relative to the case of NC. For less than 32K sockets, we get a lower efficiency compared to the case of NC (efficiency $<$ 1X). However, after 32K sockets, our scheme starts outperforming NC case ($>$ 1 efficiency values). For 340K sockets, our scheme is projected to operate the machine with an efficiency of 0.22 (95% reduction in execution time) compared to 0.01 for the NC case. Finally, for 350K sockets, the efficiency for NC case drops to 0.003 making the machine almost nonoperational using only checkpoint/restart, whereas our scheme can still operate the machine at an efficiency of 0.2.

5.2 Sensitivity to memory-per-socket and MTBF

The checkpoint time of 240 sec predicted in Kogge's report [29] is made under the assumption that an exascale machine will have 224K sockets with 64GB of memory per socket. Adding more memory to the proposed machine increases the number of components which can significantly decrease the reliability. With the proposed memory size (13.6 PB), the machine will have a flop-memory ratio of 0.01 Petaflops/TB which is far smaller than 96 Petaflops/TB and 134 Petaflops/TB for Sequoia and K computer [30] respectively. To evaluate if our scheme can enable an exascale system to have more memory, we predicted the improvement in M as well as the reduction in execution time our scheme can achieve compared to the case of NC as we keep on increasing memory per socket. Adding memory implies more data to checkpoint. We use the same methodology used in Kogge's report [29] to calculate the checkpoint time as we keep on increasing memory per socket. Figure 10 shows the results from our model for *Jacobi2D* projected on exascale machine. $MTBF$ per socket can have a significant effect on the total execution time of an application. MTBF of LANL's clusters is 10 years per socket [31] where as Jaguar had a 50 years MTBF per socket [32]. Other studies show MTBF per socket to be between 20-30 years [33, 34]. For capturing the sensitivity of our scheme to MTBF, we plotted lines corresponding to 5 different MTBF per socket ranging from 10-50 years. Figure 10 shows that our scheme will decrease the execution time for any memory size per socket using any of the five MTBF values. Even the two memory sizes used in Kogge's report [29], i.e. 16GB and 64GB, will end up benefitting from our scheme.

6. RELATED WORK

The classical solution for coping with an ever increasing failure rate due to larger machine sizes and thermal variations is to increase the checkpoint frequency. Unfortunately, checkpoint/restart can not be used indefinitely as the failure rate grows.

Some alternatives have been explored to keep up with a small $MTBF$. Using local storage to store the state of the tasks has been proposed in the double in-memory checkpoint/restart mechanism [17, 22]. Checkpointing in the memory of the nodes is fast and checkpoint periods can become smaller to tolerate frequent failures. Although this mechanism may not tolerate the failure of more than one node, several studies have confirmed that in a high percentage of the failures, only one node is affected [22, 35]. Another possibility is to improve recovery time through message-logging. In such protocols, a failure only requires the crashed node to roll back. The rest of the system will re-send the messages and wait for the crashed node to catch up with the rest of the system. A technique called parallel recovery [36] leverages message-logging by distributing the tasks on the failed node to be recovered in parallel on other nodes of the system. This mechanism has been demonstrated to tolerate a higher failure rate [37]. More recently, replication of tasks has been proposed to deal with high failure rates [38]. However, replication decreases the utilization of the system to 50% at the best. An extremely high failure rate will make

this sacrifice pay off, as the utilization of a system using checkpoint/restart drastically decreases if failures are very frequent.

In this paper we take a different approach of dealing with faults. Instead of finding efficient schemes that deal with faults, we aim to *avoid* failures by controlling temperature in all the nodes of a system using DVFS. The net result of this temperature capping is a smaller failure rate. We compensate for loss of performance due to DVFS with load balance and over-decomposition. A decreased failure rate is particularly more convenient for checkpoint/restart, but our scheme can be used in tandem with any fault-tolerance method. One of the key advantages of decreasing the failure rate is the reduction in maintenance cost of the supercomputing facility. Each failure may require at least a reboot, but in some situations manual intervention of experts is needed to diagnose the root cause of the crash.

7. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this paper provides the first study that uses runtime managed temperature capping to increase the estimated reliability of HPC machines. We formulate a model to estimate the benefits of combining checkpoint/restart and temperature capping. We first validate the accuracy of our model by comparing it with experimental data and later use it to predict the benefits of our scheme for larger machines. Experimental results show that while reducing the execution time and improving reliability, our scheme can reduce the machine energy consumption by as much as 25%. At exascale, for a 350K socket machine, regular checkpoint/restart fails to make progress (less than 1% efficiency), whereas our validated model predicts an efficiency of 20% by improving the machine reliability by a factor of up to 2.29.

Our approach uses migratability to alleviate the performance penalization induced by temperature capping. However, improving the reliability of a supercomputer also brings important benefits for environments that do not support migration, such as MPI. An immediate effect of reducing the number of failures in a system is a lower maintenance cost.

In future, we plan to investigate the effects of thermal throttling on MTBF [39] and evaluate the benefits of limiting such fluctuations. We also plan to incorporate thermal capping in other fault tolerance protocols e.g. message logging and parallel recovery [36], and compare their benefits to our current scheme.

8. ACKNOWLEDGEMENTS

This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845 and National Science Foundation under grant NSF ITR-HECURA-0833188. We are thankful to Prof. Tarek Abdelzaher for letting us use the testbed for experimentation under grant NSF CNS 09- 58314. We acknowledge the great help Xiang Ni provided us in solving several technical issues with the runtime system. We also thank Phil Miller and Shehla Saleem Rana for their valuable help in editing and proofreading the paper.

9. REFERENCES

[1] T. Renzenbrink, "Data Centers Use 1.3% of WorldÕs Total Electricity. A Decline in growth." [Online]. Available: http://www.techthefuture.com/energy/

[2] C. D. Patel, C. E. Bash, R. Sharma, M. Beitelmal, and R. Friedrich, "Smart cooling of data centers," *ASME Conference Proceedings*, vol. 2003, no. 36908b, pp. 129–137, 2003.

[3] R. F. Sullivan, "Alternating cold and hot aisles provides more reliable cooling for server farms," White Paper, Uptime Institute, 2000.

[4] R. Sawyer, "Calculating total power requirements for data centers," *White Paper, American Power Conversion*, 2004.

[5] R. American Society of Heating and A.-C. Engineers, "2008 ashrae environmental guidelines for datacom equipment." [Online]. Available: http://tc99.ashraetcs.org/documents/ASHRAE_Extended_Environmental_Envelope_Final_Aug_1_2008.pdf

[6] A. Liu, "The data center temperature debate." [Online]. Available: http://ezinearticles.com/?The-Data-Center-Temperature-Debate&id=2637938

[7] O. Sarood and L. V. Kale, "A 'cool' load balancer for parallel applications," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 21:1–21:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063412

[8] O. Sarood, P. Miller, E. Totoni, and L. V. Kale, "Cool load balancing for high performance computing data centers," vol. 61, no. 12. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 1752–1764.

[9] C. hsing Hsu, W. chun Feng, and J. S. Archuleta, "Towards efficient supercomputing: A quest for the right metric," in *In Proceedings of the HighPerformance Power-Aware Computing Workshop*, 2005.

[10] W.-c. Feng, "Making a case for efficient supercomputing," vol. 1, no. 7. New York, NY, USA: ACM, Oct. 2003, pp. 54–64. [Online]. Available: http://doi.acm.org/10.1145/957717.957772

[11] ——, "The Importance of Being Low Power in High-Performance Computing," *Cyberinfrastructure Technology Watch Quarterly (CTWatch Quarterly)*, vol. 1, no. 3, August 2005.

[12] Ericsson, "Reliability Aspects on Power Supplies," *Technical ReportDesign Note 002, Ericsson Microelectronics, April 2000.*

[13] L. Kalé, "The Chare Kernel parallel programming language and system," in *Proceedings of the International Conference on Parallel Processing*, vol. II, Aug. 1990, pp. 17–25.

[14] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, "The impact of technology scaling on lifetime reliability," in *Dependable Systems and Networks, 2004 International Conference on*, 2004, pp. 177–186.

[15] J. A. Chung H. Hsu, W. Feng, "Towards Efficient Supercomputing: A Quest for the Right Metric." [Online]. Available: http://sss.cs.vt.edu/presentations/hppac05.ppt.pdf

[16] F. Petrini, K. Davis, and J. Sancho, "System-level fault-tolerance in large-scale parallel machines with buffered coscheduling," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th*

International, 2004, pp. 209–.

[17] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *2004 IEEE International Conference on Cluster Computing*, San Diego, CA, September 2004, pp. 93–103.

[18] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Comp. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[19] J. W. Young, "A first order approximation to the optimal checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[20] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011, pp. 1 –12.

[21] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *SciDAC*, 2006.

[22] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*, 2010, pp. 1–11.

[23] "Lulesh," http://computation.llnl.gov/casc/ShockHydro/.

[24] R. K. Brunner and L. V. Kalé, "Handling application-induced load imbalance using parallel objects," in *Parallel and Distributed Computing for Symbolic and Irregular Applications*. World Scientific Publishing, 2000, pp. 167–181.

[25] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[26] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[27] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.

[28] "Intel turbo boost technology," http://www.intel.com/technology/turboboost/.

[29] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[30] "Top500 supercomputing sites," http://top500.org.

[31] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers."

[32] D. Fiala, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 2069–2072.

[33] P. Ramachandran, S. Adve, P. Bose, and J. Rivers, "Metrics for architecture-level lifetime reliability analysis," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, 2008, pp. 202–212.

[34] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 276–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006725

[35] E. Meneses, X. Ni, and L. V. Kale, "A Message-Logging Protocol for Multicore Systems," in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[36] S. Chakravorty and L. V. Kale, "A fault tolerance protocol with fast fault recovery," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[37] E. Meneses, O. Sarood, and L. V. Kale, "Assessing Energy Efficiency of Fault Tolerance Protocols for HPC Systems," in *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012)*, New York, USA, October 2012.

[38] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2011, pp. 44:1–44:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063443

[39] J. Srinivasan, S. Adve, P. Bose, and J. Rivers, "Lifetime reliability: toward an architectural solution," *Micro, IEEE*, vol. 25, no. 3, pp. 70–80, 2005.